Wydział Elektrotechniki, Automatyki i Informatyki

Informatyka stacjonarna 1 stopnia, Rok II, Semestr III

Podstawy grafiki komputerowej 1

Projekt Silnika 2D

Sprawozdanie

Do zespołu należą:

- Oliwia Kupis
- Dominik Materek
- Olivia Pacocha

Grupa: 2ID13B

Spis treści

1. Obsługa klawiatury i myszy (wykonał: Dominik Materek)

Poprzez dodanie w nagłówku #include <SFML/Window/Keyboard.hpp>, możemy nadać dowolnemu przyciskowi na klawiaturze funkcjonalności.
Przycisk zapisujemy jako:

sf::Keyboard::NAZWAPRZYCISKU

Wciśnięty przycisk zapisujemy jako:

sf::Keyboard::isKeyPressed(sf::Keyboard::NAZWAPRZYCISKU)

Poprzez dodanie w nagłówku #include <SFML/Window/Mouse.hpp>, możemy nadać dowolnemu przyciskowi na myszce funkcjonalności.
Przycisk zapisujemy jako:

sf::Mouse::NAZWAPRZYCISKU

Wciśnięty przycisk zapisujemy jako:

sf::Mouse::isButtonPressed(sf::Mouse::NAZWAPRZYCISKU)

Obsługa klawiatury oraz myszy znajduje się w klasie Engine, w metodzie Engine::processEvents(). Dzięki temu silnik rozpoznaje, który przycisk został naciśnięty oraz zwolniony na klawiaturze lub myszce oraz rozpoznaje pozycje myszy na ekranie.

Przykład użycia przycisków klawiatury w klasie Player.cpp:

```
void Player::update(const sf::RenderWindow& window) {
    float dt = 1.f / 60.f;

    sf::Vector2f movement(0, 0);
    bool moved = false;
    Direction newDir = currentDirection;

    // sterowanie
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::W)) {
        movement.y -= _speed * dt;
        newDir = Up;
        moved = true;

    } else if (sf::Keyboard::isKeyPressed(sf::Keyboard::S)) {
        movement.y += _speed * dt;
        newDir = Down;
        moved = true;
    }

    if (sf::Keyboard::isKeyPressed(sf::Keyboard::A)) {
        movement.x -= _speed * dt;
        newDir = Left;
        moved = true;

    } else if (sf::Keyboard::isKeyPressed(sf::Keyboard::D)) {
        movement.x += _speed * dt;
        newDir = Right;
        moved = true;
    }
}
```

2. Obsługa współrzędnych Point2D (wykonała: Oliwia Kupis)

Klasa Point2D reprezentuje punkt w dwuwymiarowej przestrzeni, przechowuje jego współrzędne X i Y, a także oferuje metody do ich uzyskania, ustawiania i rysowania za pomocą PrimitiveRenderer. Przykładem użycia może być rysowanie pojedynczego punktu lub na przykład narysowanie kilku punktów w linii w metodzie render klasy Engine.

Rysowanie punktu:

```
void Engine::render() { _
    window.clear(_clearColor);
    PrimitiveRenderer renderer(_window);
    Point2D p1(100, 100);
    p1.draw(renderer, sf::Color::Yellow);
    _window.display();
}
```

Rysowanie kilku punktów w linii:

```
void Engine::render() { _
    window.clear(_clearColor);
    PrimitiveRenderer renderer(_window);

    for (int i = 0; i < 10; ++i) {
       Point2D p(50 + i * 10, 200);
       p.draw(renderer, sf::Color::Green);
    }
    _window.display();
}
```

3. Rysowanie prymitywów (wykonały: Oliwia Kupis, Olivia Pacocha)

a) Rysowanie odcinka algorytmem przyrostowym

Metoda do rysowania odcinka algorytmem przyrostowym to drawLineInctremental(plik PrimitiveRenderer.cpp). Aby ją narysować musi zostać wywołana w metodzie render klasy Engine w pliku Engine.cpp z parametrami: początek linii (x,y), koniec linii (x,y), true/false czy ma być algorytm przyrostowy, kolor linii.

```
void Engine::render() { _
    window.clear(_clearColor);
    PrimitiveRenderer renderer(_window);
    LineSegment lineB({ 400, 100 }, { 500, 250 });
    lineB.draw(renderer, true, sf::Color::Red);
    _window.display();
}
```

b) Rysowanie linii łamanej otwartej lub zamkniętej

Metoda do rysowania linii łamanej otwartej lub zamkniętej to drawPolyLine(plik PrimitiveRenderer.cpp). Aby ją narysować musi zostać wywołana w metodzie render klasy Engine w pliku Engine.cpp z parametrami: każdy kolejny punkt (x,y) po przecinku, kolor linii, true/false dla linii zamkniętej lub otwartej.

```
void Engine::render() { _
    window.clear(_clearColor);
    PrimitiveRenderer renderer(_window);
    std::vector<Point2D> poly = { {50,300}, {100,350}, {150,300},
    {200,350} };
    renderer.drawPolyline(poly, sf::Color::Magenta, false);
    _window.display();
}
```

c) Rysowanie okręgu 8-krotną symetrią

Metoda do rysowania okręgu to drawCircleSummetry w klasie PrimitiveRenderer (plik PrimitiveRenderer.cpp). Aby narysować okrąg musisz wywołać ją w metodzie render przy pomocy zmiennej lokalnej (tu: renderer) klasy Engine w pliku Engine.cpp z parametrami: środek okręgu {x,y}, promień koła oraz kolor krawędzi.

```
void Engine::render() { _
    window.clear(_clearColor);
    PrimitiveRenderer renderer(_window);

    renderer.drawCircleSymmetry({ 800, 100 }, 50, sf::Color::White);
    _window.display();

}
```

d) Rysowanie elipsy 8-krotną symetrią

Metoda do rysowania elipsy to drawEllipseSummetry w klasie PrimitiveRenderer (plik PrimitiveRenderer.cpp). Aby narysować elipsę należy wywołać ją w metodzie render przy pomocy zmiennej lokalnej klasy Engine w pliku Engine.cpp z parametrami: środek elipsy {x,y}, długość promienia w szerz, długość promienia w wzdłuż oraz kolor krawędzi.

```
void Engine::render() { _
    window.clear(_clearColor);
    PrimitiveRenderer renderer(_window);

    renderer.drawEllipseSymmetry({800,225}, 50, 30, sf::Color::White);

    _window.display();

}
```

e) Rysowanie dowolnego wielokąta

Metoda PrimitiveRenderer::drawPolygon rysuje wielokąt na podstawie podanych wierzchołków, sprawdzając jednocześnie, czy krawędzie nie przecinają się, aby uniknąć błędów w rysowaniu. Aby ją wywołać należy wywołać je w metodzie render klasy Engine przy pomocy zmiennej renderer podając wierzchołki i kolor krawędzi.

```
void Engine::render() { _
    window.clear(_clearColor);

    PrimitiveRenderer renderer(_window);

    std::vector<Point2D> figure1 = {
    {700, 500}, {800, 600}, {750, 400};
    renderer.drawPolygon(figure1, sf::Color::White);

    _window.display();

}
```

4. Wypełnianie prymitywów kolorem (wykonała: Olivia Pacocha)

a) Rysowanie prymitywów wypełnionych kolorem

Metoda drawFilledPolygon klasy PrimitiveRenderer rysuje wypełniony wielokąt na podstawie podanych wierzchołków i koloru. Metoda drawFilledCircle klasy PrimitiveRenderer rysuje wypełnione koło na oknie, przyjmując jako argumenty środek koła, promień oraz kolor wypełnienia. Należy je wywołać w metodzie render klasy Engine przy pomocy zmiennej renderer.

```
void Engine::render() { _
    window.clear(_clearColor);
    PrimitiveRenderer renderer(_window);
```

```
    std::vector<sf::Point2D> figure = {
     {800, 500}, { 900,500 }, { 900,600}, {800, 600} };

    renderer.drawFilledPolygon(figure, sf::Color::White);

    renderer.drawFilledCircle({ 1000, 550 }, 50, sf::Color::White);
    _window.display();
}
```

b) Wypełnianie obszaru kolorem przez spójność (border fill, flood fill)

Metoda borderfill wypełnia obszar w oknie graficznym kolorem, zaczynając od podanego punktu, aż do napotkania koloru granicy, wykorzystując algorytm wypełniania obszaru. Metoda floodfill klasy PrimitiveRenderer wypełnia obszar na podstawie podanego punktu startowego i koloru, wykorzystując algorytm przeszukiwania w głąb do zmiany koloru pikseli w obrębie obszaru o tym samym kolorze co piksel startowy. Należy je wywołać w metodzie render klasy Engine przy pomocy zmiennej renderer.

```
void Engine::render() { _
    window.clear(_clearColor);

//border fill
    std::vector<sf::Point2D> figure1 = {
    {800, 650}, { 900,650 }, { 900,750}, {800, 750} };
    renderer.drawPolygon(figure1, sf::Color::White);

    renderer.borderfill(
        {850,700},
        sf::Color(200,240,255),
        sf::Color::White);

//flood fill
    std::vector<sf::Point2D> figure2 = {
    {650, 650}, { 750,650 }, { 750,750}, {650, 750} };
    renderer.drawPolygon(figure2, sf::Color::White);

    renderer.floodfill({ 700, 700 }, sf::Color(255, 200, 230));
    _window.display();

}
```

5. Przekształcenia geometryczne (wykonał: Dominik Materek)

CircleObject to klasa, która reprezentuje okrąg zdefiniowany przez środek i promień; używa metody `drawCircleSymmetry` do rysowania z wykorzystaniem 8-krotnej symetrii, umożliwia zmianę położenia, koloru oraz promienia i integruje się z hierarchią obiektów dziedzicząc po `GameObject`.

Parametry:

- `sf::Vector2f _center` – współrzędne środka okręgu
- `float _radius` – promień okręgu
- `sf::Color _color` – kolor linii okręgu

EllipseObject to klasa reprezentująca elipsę (lub okrąg), której rysowanie odbywa się w Engine::render() poprzez wywołanie PrimitiveRenderer::drawCircleSymmetry(center, radius, color), wykorzystującego 8-krotną symetrię do szybkiego rysowania.

Parametry:

- float x – środek X
- float y – środek Y
- float rx – promień poziomy
- float ry – promień pionowy
- sf::Color color – kolor

LineObject to klasa reprezentująca odcinek, który jest rysowany w Engine::render() za pomocą funkcji PrimitiveRenderer::drawLine(p1, p2, color), wykorzystującej własny algorytm rasteryzacji linii

Parametry:

- float x1, y1 – punkt początkowy
- float x2, y2 – punkt końcowy
- sf::Color color – kolor linii

PointObject to klasa reprezentująca pojedynczy punkt, który jest rysowany w Engine::render() przy użyciu funkcji PrimitiveRenderer::drawPixel(position, color), zapisującej jeden piksel w buforze ekranu.

Parametry:

- float x – współrzędna X
- float y – współrzędna Y
- sf::Color color – kolor punktu

PolygonObject to klasa reprezentująca wielokąt złożony z wielu punktów, rysowany w Engine::render() przez wywołanie PrimitiveRenderer::drawLine() dla każdego kolejnego boku, łączącego wszystkie wierzchołki oraz opcjonalnie zamykającego kształt.

Parametry:

- std::vector<sf::Vector2f> points – lista wierzchołków
- sf::Color color – kolor

PolylineObject to klasa reprezentująca linię łamaną, rysowaną w Engine::render() poprzez wywoływanie PrimitiveRenderer::drawLine() dla kolejnych par punktów, bez automatycznego zamykania kształtu.

Parametry:

- std::vector<sf::Vector2f> points – lista punktów
- sf::Color color – kolor

RectangleObject to klasa reprezentująca prostokąt, rysowany w Engine::render() poprzez wywołanie PrimitiveRenderer::drawLine() dla jego czterech krawędzi lub za pomocą własnej funkcji rysowania prostokąta.

Parametry:

- float x – pozycja środka lub lewego góry (zależnie od implementacji)
- float y
- float width – szerokość
- float height – wysokość
- sf::Color color – kolor obramowania

TriangleObject to klasa reprezentująca trójkąt, rysowany w Engine::render() przez wywołanie PrimitiveRenderer::drawLine() dla trzech boków łączących wszystkie jego wierzchołki.

Parametry:

- sf::Vector2f p1 – pierwszy wierzchołek
- sf::Vector2f p2 – drugi wierzchołek
- sf::Vector2f p3 – trzeci wierzchołek
- sf::Color color – kolor

Metody manipulacji obiektami:

.draw(sf::RenderWindow& window):

By obiekt pojawił się na ekranie użyj metody .draw(sf::RenderWindow& window), gdzie zamiast "sf::RenderWindow& window" wpisujesz nazwę okna w którym chcesz by pojawił się obiekt.

.translate(float dx, float dy):

By przesunąć obiekt użyj metody .translate(float dx, float dy), gdzie zamiast "float dx" oraz "float dy" wpisujesz współrzędne przesunięcia.

.rotate (float angle):

By obrócić obiekt użyj metody .rotate(float angle), gdzie zamiast "float angle" wpisujesz kąt o jaki chcesz obrócić obiekt.

.scale(float factorX, float factorY):

By przeskalować obiekt użyj metody .scale(float factorX, float factorY), gdzie zamiast "float factorX" wpisujesz o ile chcesz przeskalować obiekt względem osi X oraz "float factorY" wpisujesz o ile chcesz przeskalować obiekt względem osi Y.

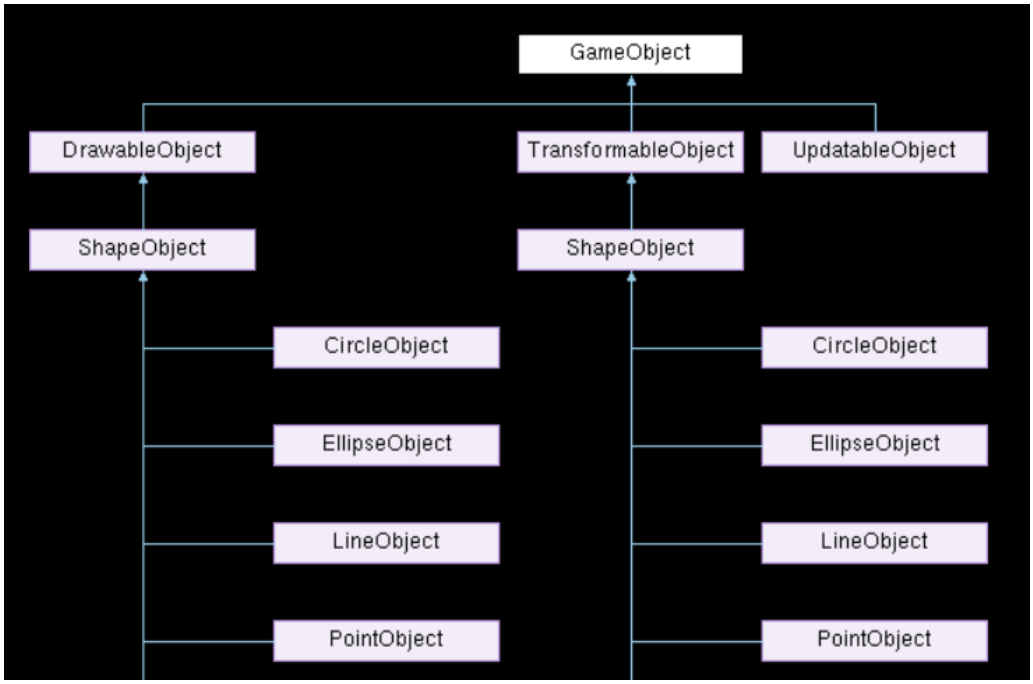
Przykład użycia tych metod:

```
void Engine::render() { _
    window.clear(_clearColor);
      PolylineObject polyline({
      {100, 200},
      {150, 150},
      {200, 200},
      {250, 180}
      }, sf::Color::Cyan); //stworzenie obiektu linii łamanej

      polyline.rotate(30); //rotacja
      polyline.scale(1.2f, 1.f); //skala
      polyline.translate(50, 40); //przesunięcie
      polyline.draw(_window); //narysowanie na ekranie

      _window.display();

}
```

6. Hierarchia klas (wykonał: Dominik Materek)

Hierarchia klas polega na wirtualnym dziedziczeniu.

Schemat:



GameObject- bazowa klasa każdego obiektu w grze, posiada ID i podstawowy interfejs do aktualizacji oraz renderowania.

UpdatableObject (dziedziczy po GameObject)- obiekt, który posiada logikę aktualizacji w czasie, definiuje metodę update(dt).

DrawableObject (dziedziczy po GameObject)- obiekt, który może zostać narysowany, definiuje metodę draw(renderer).

TransformableObject (łączy cechy updatable + drawable) - o biekt, który można przesuwać, skalować i obracać, zawiera transformacje do wykorzystania przy rysowaniu.

ShapeObject (bazowa klasa prymitywów geometrycznych- abstrakcyjna klasa dla wszystkich figur, udostępnia wspólny interfejs do transformacji oraz koloru.

7. Obsługa bitmap (wykonała: Oliwia Kupis)

Do obsługi bitmap używana jest klasa BitmapHandler w pliku BitmapHandler.h. Klasa ta umożliwia nam inicjalizacje oraz zwalnianie zasobów zajmowanych przez obiekt BitmapHandler, ładowanie bitmap z pliku, zapisywanie bitmap do pliku, kopiowanie całości bitmapy lub jej fragmentu, modyfikacje pikseli, konwersje do tekstury, a także obsługi kanału alfa. Natomiast istnieje także klasa bazowa dla obiektów bitmapowych czyli BitmapObject w pliku BitmapObject.h odpowiadająca za przekształcanie tekstury w sprite'y oraz ich rysowanie.

8. Animowanie bitmap (wykonała: Oliwia Kupis)

Do obsługi animacji bitmap wykorzystujemy klasę AnimatedObject w pliku AnimatedObject.h oraz SpriteObject w pliku SpriteObject.h. Klasa AnimatedObject przechowuje wirtualny destruktor, aby zapewnić poprawne usuwanie obiektów pochodnych oraz metodę animate animującą obiekt z parametrem dt, czyli czas jaki upłynął od ostaniej animacji. Implementacja metody animate jest wykorzystywana w klasie SpriteObject, w tym miejscu implementowany jest czas trwania pojedynczej klatki oraz ewentualne wykorzystywanie bitmapy w formie fragmentu.

```
virtual void animate(float dt) override {
    time += dt;
    if (time >= frameTime) {
        time = 0;
        currentFrame = (currentFrame + 1) % frames;

        sprite.setTextureRect({
            currentFrame * frameWidth,
            currentDirection * frameHeight,
            frameWidth,
            frameHeight
            });
    }
}
```

Natomiast klasa SpriteObject odpowiada za całą animację bitmap.
Metoda :

- setSpriteSheet ustawia arkusz sprite'ów oraz parametry ich animacji,
- draw rysuje obiekt na oknie,
- translate przesuwa obiekt o wartości dx i dy jako parametry,
- rotate obraca obiekt o określony kąt jako parametr angle,
- scale skaluje obiekt o wartości sx i sy jako parametry,
- setDirection ustawia kierunek animacji.

Zmienne:

- frameWidth - szerokość pojedynczej klatki,
- frameHeight - wysokość pojedynczej klatki,
- frames – liczba klatek w animacji,
- currentFrame – aktualna klatka animacji,
- currentDirection – aktualny kierunek animacji
  (0 = dół, 1 = lewo, 2 = prawo, 3 = góra),
- time – akumulator czasu do animacji,
- frameTime – czas trwania pojedynczej animacji klatki.

Wszystkie te metody i zmienne są implementowane w klasie Player pliku Player.h, dodatkowo został stworzony konstruktor gracza jako Player z argumentami (x,y,size), gdzie x i y to początkowa pozycja gracza, a size to rozmiar.

Konstruktor ten natomiast jest implementowany w metodzie run klasy Engine.

Wykorzystanie metod i zmiennych w Player.h:

```
class Player : public SpriteObject {
public:
    enum Direction { Down = 0, Left = 1, Right = 2, Up = 3 };

    Player(float x = 100.f, float y = 100.f, float size = 40.f);

    void draw(sf::RenderWindow& window) override;

    void translate(float dx, float dy) override;

    void rotate(float angle) override;

    void scale(float sx, float sy) override;

    void update(const sf::RenderWindow& window);

private:

    bool loadFrame(const std::string& path, Direction dir, int frameIndex);

    float _speed = 200.f;

    std::array<std::vector<std::shared_ptr<sf::Texture>>, 4> textures;

    int framesPerDir = 3;

    Direction currentDirection = Down;

    int currentFrame = 1;

    float timeAccumulator = 0.f;

    float animationFPS = 10.f;

    bool isMoving = false;

    int frameWidth = 32;

    int frameHeight = 32;

    void applyCurrentFrame();
};
```